Simulation Modeling and Statistical Computing

Richard E. Nance Editor

Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks

BORIS D. LUBACHEVSKY

ABSTRACT: Simulating asynchronous multiple-loop networks is commonly considered a difficult task for parallel programming. Two examples of asynchronous multiple-loop networks are presented in this article: a stylized queuing system and an Ising model. In both cases, the network is an $n \times n$ grid on a torus and includes at least an order of n^2 feedback loops. A new distributed simulation algorithm is demonstrated on these two examples. The algorithm combines three elements: (1) the bounded lag restriction; (2) minimum propagation delays; and (3) the so-called opaque periods. We prove that if N processing elements (PEs) execute the algorithm in parallel and the simulated system exhibits sufficient density of events, then, on average, processing one event would require $O(\log N)$ instructions of one PE. Experiments on a shared memory MIMD bus computer (Sequent's Balance) and on a SIMD computer (Connection Machine) show speed-ups greater than 16 on 25 PEs of a Balance and greater than 1900 on 2^{14} PEs of a Connection Machine.

1. INTRODUCTION

The queuing network paradigm has been widely used in distributed event-driven simulations [3, 4, 6, 7, 9, 10, 11, 17, 22, 23]. The ease with which a system can be described in the form of a queuing network does not necessarily imply a comparable ease in simulating the system in this form. Assuming that different queues are hosted by different processing elements (PEs), when two queues are not directly connected in the network, it does not follow that the two PEs hosting these queues are prohibited from direct communication with each other. However, many queuing-network-style algorithms silently assume that if the algorithm is not centralized, its topology of communication must be in a strict agreement with the topology of communication in the simulated network.

The unnecessary restriction of topological isomorphism can make the algorithm unduly sensitive to the topology of the simulated network. These algorithms usually efficiently simulate a straight line of queues (Fig. 1*a*). However, unless roll-back is allowed [7, 11], the deadlock problem emerges as soon as alternative paths (Fig. 1*b*) appear in the network. The means to counter deadlock include: infinite buffers [10, 22], additional messages [10, 17], and a detection/resolution method [4]. The problem of deadlock may become harder to solve when the simulated network has feedback loops (Fig. 1*c*). In this case even large buffers do not prevent deadlock [22], and using either of the other two approaches may result in low speed-up as shown empirically [6, 23]. It is commonly believed that each additional loop on the network graph significantly degrades simulation performance. Proposed topology-insensitive algorithms are accompanied by other difficulties such as the danger of cascading roll-backs [7, 11], or a low degree of achievable parallelism [12].

From the literature one might get the impression that a multiple-loop asynchronous network is intractable for an efficient distributed simulation. This article attempts to change the impression by presenting two examples of efficient distributed simulation of such networks. One example, the token transport network, is a stylized asynchronous queuing system, while the other, an asynchronous Ising system, belongs to computational physics. These are case presentations of a new distributed event-driven algorithm.

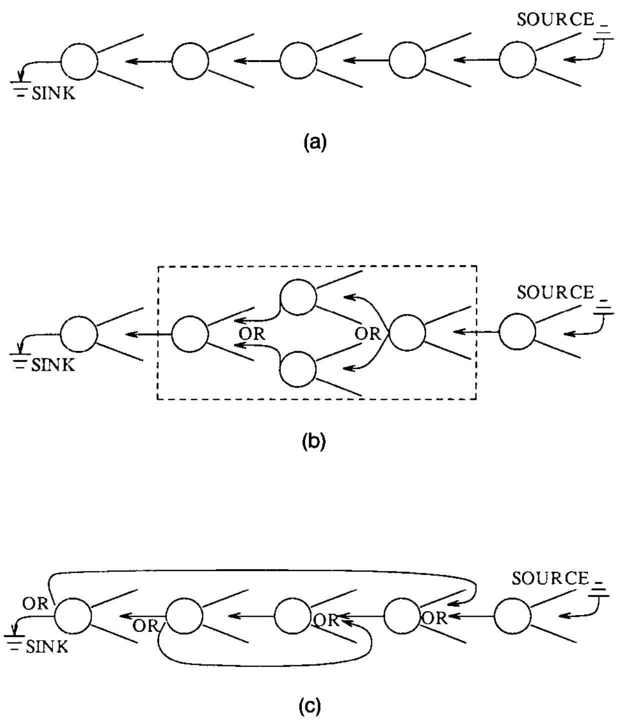


FIGURE 1. Network Topologies

Described in an abstract form unrelated to queues or other specific applications, the algorithm avoids unnecessary restricutions inherited from an application. The stylized queuing example shows how an application can be mapped into this abstract form. The main elements of the algorithm, which are easily identified in this form, are: (1) the *bounded lag*, which means that the difference in the simulated time between events being processed concurrently is bounded from above by a known finite constant; (2) *minimum propagation delays* between the parts of the simulated system; and (3) the so-called *opaque periods*, which are the delays caused by the non-preemptive states these parts can enter, as explained in Section 3.

In general, the algorithm has a synchronous structure proceeding in iterations, while the simulated system may be asynchronous. In specific cases, like the Ising simulation, the synchrony can be relaxed. Since there is no deadlock or blocking in the algorithm, there is no need for deadlock detection/resolution or avoidance. Each iteration is guaranteed to terminate with some events having been processed at it.

The combined use of the previously mentioned three elements in one algorithm is new, as is the way these elements are combined; however, the use of any one element has been reported in the literature. Specifically, in those queuing system models that assume infinite buffers and zero transmission durations, both the idea of a minimum-propagation delay and the idea of an opaque-period stem from the phenomenon of a nonzero service duration. In such contexts, these two elements have been recognized since the early algorithms [3, 10, 22]. However, in the complex environment of a specific example, the difference between opaque periods and propagation delays can be masked. This difference does not seem to be clearly stated in literature. Demonstrating the difference is one purpose of presenting the Ising simulation example, where the precomputed minimum propagation delays degenerate to zero, but substantial opaque periods still exist.

Recently, the idea of the bounded lag was independently introduced as a *moving time window* [24]. Its implementation [24] is different from that proposed here. Except for the case where the window size is so small that it cannot contain two causally connected events processed by different PEs, the algorithm [24] guarantees no correctness in the usual sense or even reproducibility of the simulation. In contrast, correctness and reproducibility in our algorithm are preserved for any nonnegative lag value.

Many authors appreciate the difficulty of a theoretical performance evaluation for an asynchronous distributed discrete event simulation and limit themselves to empirical evaluation. For example, survey [17] proposes "... the empirical investigation of various heuristics on a wide variety of problems ...." In contrast, this article presents an *efficiency proof* for the proposed algorithm. We show that on an appropriate physically realizable massively parallel processor, the algorithm is at least an order of $1/\log N$ scalable. This means that if the sizes of both the simulated system and the simulator increase proportionally, keeping local properties intact, then the order of the ratio of progress rates of processing time over simulated time increases as $O(\log(\text{size}))$. Specifically, in our queuing network example, if N PEs execute the algorithm in parallel, so that each queue is hosted by its own PE, and the simulated system exhibits sufficient density of events so that at each iteration there are, on average, an order of N nonempty queues with pending events within the bounded lag horizon, then the average of $O(\log N)$ instructions of one PE will suffice for processing one event. Of this cost, only $O(1)$ instructions per event takes proper event processing. The algorithm synchronization accounts for the rest of the cost. Best serial list manipulation techniques, that do not use constructs 1–3 above, require the average of an order $\log N$ instructions for processing one event. Since the cost per event in the parallel algorithm is asymptotically not less than in the serial algorithm, but N times more processors execute the parallel algorithm, the speed-up is provably not less than of an order of N, i.e., *linear speed-up*.

The proposed simulation algorithm is intended for parallel execution, however, it can also be run serially. A number of such serial experiments were performed for the queuing system example. Their results clearly demonstrate the potential for a high parallel speed-up. A repetition of these experiments for a parallel computer is currently in progress. Their preliminary results confirm the prediction of a high speed-up.

For the Ising model, simulation experiments were also performed on the available parallel hardware which was a Connection Machine® and a Balance® computer. The efficiencies and speed-ups in these runs met predictions and were substantial. For example, the observed speed-up on 25 PEs of the Balance computer was greater than 16 while on 2^{14} PEs of the Connection Machine the speed-up was greater than 1900.

The paper is organized as follows: First, the two multiple-loop network examples are presented together with their mapping into an abstract world view required by the algorithm. Three algorithm elements are defined and identified in the examples. The algorithms are then described, and the results of computer experiments and a performance evaluation, including an outline of the performance proof follow. We conclude by listing other feasible applications of the algorithm, its known shortcomings, and directions for further research.

2. EXAMPLES

A Token Transport Network

The simulated system is an $n \times n$ toroidal network. (A 4×4 example is depicted in Figure 2.) A node of the network is a server with an attached input queue buffer of infinite capacity. An *idling* server is constantly trying to change its status to *serving* by removing a token from the input buffer and starting service. The service durations are bounded from below by σ seconds, $\sigma > 0$. When the service is completed, the token is either deleted from the system or dropped into the input queue of one of the four neighboring servers. After a token is served, the server assumes an *idling* status if no more tokens remain in its input queue or, if at least one token is left in the input queue, the server removes one token from the queue and resumes a *serving* status. A server which always deletes a token can be viewed as a "sink." A server whose input buffer is full to capacity $(+\infty)$ can be viewed as a "source."

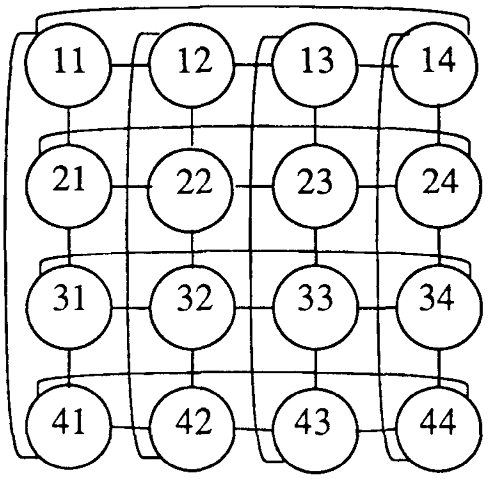


FIGURE 2. A 4 × 4 Toroidal Network

In a distributed simulation algorithm, a server can be represented by a separate PE. PEi, which hosts simulation on behalf of server i, maintains the *pool of pending local events* Π_i. Each event $e \in \Pi_i$ is a pair $e =$ (contents, time). PEi processes its events one at a time, during which it modifies its own pool Π_i, and/or the pools Π_j of its neighbors (in the simulated network). A detailed description of this scheme follows.

The state of server i is a pair (s_i, q_i), where s_i indicates the server status, that is, $s_i =$ *idling* or $s_i =$ *serving*, and $q_i \geq 0$ is the integer number of tokens contained by the input buffer. There are three types of events: (1) begin service; (2) end service; and (3) drop a token into the input queue. The contents of an event consist of the event type and the node number where the event occurs. For example, $(3, i, t)$ is an event with contents $(3, i)$ and time t. This event means "drop a token into the input queue of server i at time t" and it belongs to pool Π_i.

Five rules for local event processing are presented in Figure 3. Each rule tells what PEi must do to process a particular event e, provided that: (1) this event e is chosen for processing; and (2) a specified condition holds. For example, rule 5 describes the processing of event $(3, i, t)$, if $s_i =$ *serving*.

Simulation begins with some pools Π_i being non-empty. Each non-empty pool must contain exactly one event $(1, i, 0)$ and the buffer must be non-empty: $q_i > 0$. The corresponding server begins service as soon as the

```
Rule 1. Process event $(1, i, t)$, if $q_i > 0$, as follows:
  1.1. Determine:
        a) the service duration $t_{service} \geq \sigma$;
        b) whether or not the token is deleted after the service;
           and
        c) which output $j$ receives the token, if any
  1.2. Change state $s_i :=$ serving and $q_i := q_i - 1$
  1.3. Insert event $(2, i, t + t_{service})$
  1.4. If the token is not deleted, then insert event
        $(3, j, t + t_{service})$
Rule 2. Process event $(2, i, t)$, if $q_i > 0$, as follows:
  Insert event $(1, i, t)$
Rule 3. Process event $(2, i, t)$, if $q_i = 0$, as follows:
  Change state $s_i :=$ idling
Rule 4. Process event $(3, i, t)$, if $q_i = 0$ and $s_i =$ idling, as
follows:
  4.1. Change state $s_i :=$ serving and $q_i := q_i + 1$
  4.2. Insert event $(1, i, t)$
Rule 5. Process event $(3, i, t)$, if $s_i =$ serving, as follows:
  Change state $q_i := q_i + 1$
```

FIGURE 3. Rules for Processing Events

® Connection Machine is a registered trademark of Thinking Machines Corporation.
® Balance is a trademark of Sequent Computer Systems, Inc.

simulation starts at $t = 0$. Each empty pool Π_i must, at the start, have an empty buffer: $q_i = 0$.

The following two situations are not covered in Figure 3: (1) processing $(1, i, t)$, if $q_i = 0$; and (2) processing $(3, i, t)$, if $s_i = idling$ and $q_i > 0$. Starting from the initial condition, one can formally prove that neither (1) nor (2) can occur during simulation. This conforms with intuition; a service start cannot be processed if there is nothing to serve, and a server cannot be *idling* while there are tokens in its buffer.

Instruction 1.1 in Figure 3 commands the PE to determine $t_{service}$, decide whether or not the token is deleted, and determine the output j which receives the token if the token is not deleted. These determinations can be arbitrary, within the given restrictions; i.e., $t_{service} \geq \sigma$ and j must be a neighbor of i. For concreteness, in the computer experiments reported in Section 5, these determinations are as follows:

1. Service durations $t_{service}$ are independent realizations of a random variable, $t_{service} = \sigma + \zeta$, where σ is a constant, variable ζ is distributed exponentially, and $E[\zeta] = \mu - \sigma$, so that $E[t_{service}] = \mu$.
2. The token is deleted after the service with probability p; this deletion is independent of any other decision.
3. The choice of which output j should receive the token is made randomly and uniformly among the four neighbors of i; this decision is independent of any other.

In the preceding statements, σ, μ, and p are fixed parameters, $0 < \sigma \leq \mu$, $0 \leq p \leq 1$.

The rules in Figure 3 do not tell how events are chosen for processing. This choice should be determined by a general strategy of simulation. A conventional strategy maintains a single event-list, where the events are ordered according to their times. An event or events with minimum times are chosen for processing, and before this processing is completed, no other event can be processed. This strategy should be qualified as serial, even if the simulated system is distributed over the different PEs of a parallel processor. The potential for a speed-up while using this strategy is limited because a single-event-list processing bottlenecks a sufficiently large system.[1] This article describes a different, truly distributed strategy of simulation.

Asynchronous Ising Model

In the model of an Ising spin system, the same $n \times n$ torus network that was used in the queuing system example hosts a physical atom at each node. The atom, located at node i, has a magnetic spin $s(i)$: $s(i) = +1$ or $s(i) = -1$. In the asynchronous version of the Ising model [8], the spins attempt to change at asynchronous, discrete times. Attempted spin change arrivals for a particular atom are random and form a Poisson process. Arrivals for different atoms are independent, the arrival rate is the same, say λ, for each atom.

When an attempt arrives, the new spin is determined, using the spin values of the given atom and the neighboring atoms just before the update time. A random experiment ω may be also involved in the determination:

$$s_t(i) := new\_state(s_{t-0}(neighbors(i)), \omega). \qquad (2.1)$$

In (2.1), $s(neighbors(i))$ denotes the indexed set of states of all the neighbors of i including i itself. Subscript $t - 0$ expresses the idea of "just before t." According to (2.1), the value of $s(i)$ instantaneously changes at time t from $s_{t-0}(i)$ to $s_t(i)$. At time t, the value of $s(i)$ is already new. Concrete functions *new_state*( ) in (2.1) are given in the literature on the Ising model (e.g., [1]; see also [14]), and are not important for the purposes of this article.

In this example, there is only one event type, "an attempted spin change," and only one processing rule, formula (2.1).

A well-known standard algorithm to simulate this system was invented by Metropolis et al. [16]. In the standard algorithm, the evolution of the spin configuration is a sequence of one-spin updates: Given a configuration, define the next configuration by choosing a node i uniformly at random and attempting to change the spin $s(i)$ according to formula (2.1).

The standard algorithm can be considered as a variant of the conventional single event-list technique, although no event-list is explicitly maintained. Here, N separate event streams are replaced by a single cumulative event stream and by a procedure which randomly delegates a node to meet each arrival. Thus, the time increment between consecutive spin change attempts is an independent realization of an exponentially distributed random variable with mean $1/(\lambda N)$, where $N = n^2$ is the total number of atoms in the network. This simplification becomes possible because a sum of independent Poisson streams is a Poisson stream. If the arrival streams for different atoms do not enjoy this additivity property, an explicit event-list, one for all atoms, could be the best strategy [13]. With or without the event-list, the strategy is serial. This article presents efficient distributed strategies for simulating this model.

3. ELEMENTS OF THE ALGORITHM

Bounded lag

Let $\tau(e)$ denote the time of event e. The *bounded lag restriction with parameter B* is:

> If events e_1 and e_2 are processed concurrently then $|\tau(e_1) - \tau(e_2)| \leq B$, where $0 \leq B < +\infty$ is a known constant.

To maintain the bounded lag restriction, the *(simulation) floor* equal

$$\min_{i=1,2,\ldots N, e \in \Pi_i} \tau(e)$$

[1] Possibilities to parallelize this strategy are explored in [12], which concludes that at most speed-up of an order of log N can be achieved, but only when processing one event generates many new pending events.

is computed. An event e is admitted for processing only if $\tau(e) \leq floor + B$.

The Minimum Propagation Delay

Let a graph G be associated with the simulated system. The nodes of G, numbered $1, \ldots, N$, identify the N parts of the system. G has a directed link $i \rightarrow j$, if (1) processing of an event e, $e \in \Pi_i$, can change pool Π_j or (2) the state of node i can directly influence the history at node j. Pool Π_j in (1) can be changed in several ways: new events can be scheduled at node j, or events already scheduled at node j can be canceled, or both.

For the preceding examples, G coincides with the graph of the $n \times n$ toroidal network and $N = n^2$. An example of item (1) in the definition above is instruction 1.4 in Figure 3: event $(3, j, t + t_{service})$ is inserted into Π_j while processing event $(1, i, t) \in \Pi_i$. Rule (2.1) for processing spin change attempts is an example of item (2). Note that if the state of a node is augmented with states of its neighbors, then (2) can be reduced to (1). This transformation would result in a more tedious though equivalent model for the Ising simulation.

Before the simulation starts, the *minimum propagation delay* $d(i, j) \geq 0$ is assigned to each ordered pair (i, j) of nodes of G. The assignment is carried out in the following steps: (1) $d(i, i) = 0$ for all i; (2) if $i \rightarrow j$ is a link in G, then an analysis of event processing rules yields the largest estimate $d(i, j)$ such that an event or a state change which occurs at time t at node i can affect the history at node j no earlier than at time $t + d(i, j)$; (3) if there exists a unique directed path from i to j, then $d(i, j)$ is the sum of the delays along this path; if there are several such paths, $d(i, j)$ is the minimum sum over all such paths; (4) if there is no directed path from i to j, then $d(i, j) \overset{def}{=} +\infty$.

With this definition the triangle inequality, $d(i, j) + d(j, k) \geq d(i, k)$, is guaranteed for any three nodes i, j, and k.

Consider the example of a queuing system. Since $t_{service} \geq \sigma$ by rule 1 in Figure 3, minimum delay $d(i, j)$ equals σ for any two neighbors i and j. For the pairs i and j which are not neighbors, $d(i, j)$ is set equal to σ times the number of links on a shortest path from i to j. For example, $d(11, 44) = 2\sigma$ in Figure 2 since $(11) \rightarrow (14) \rightarrow (44)$ is a shortest path from (11) to (44). Note that $d(i, j) = d(j, i)$. Although this symmetry is convenient, it is not required in the general case.

In this example, definitions 1–4 are equivalent to the following abstract definition: *$d(i, j)$ is the largest lower bound on the delay for the history of states at node j to become affected by a change at node i, that can be found before the simulation starts without knowing the change or the states.* Unfortunately, the latter definition is not equivalent to the former in a general case. We believe that a good mapping of a particular system into the structure of the distributed simulation algorithm is always possible for which both definitions are equivalent. An instance of bad mapping: a three-noded network $i \rightarrow m \rightarrow j$, where $d(i, m) = d(i, j) = 1$, but no change at i can affect the history at j, because m consists of two separate parts, m_1 and m_2, where m_1 communicates only with i and m_2 communicates only with j. For $d(i, j)$ defined according to (3) as $d(i, j) = d(i, m) + d(m, j) = 2$, the property of $d(i, j)$ contained in the abstract definition does not hold, since changes at i never propagate to j. A better mapping splits m into two distinct nodes m_1 and m_2. Then $d(i, m_1) = d(m_2, j) = 1$, $d(m_1, m_2) = d(i, j) = +\infty$, and the two definitions coincide.

Given a constant $r > 0$ and a node i, the *incoming spherical region* $S \downarrow (i, r)$, with its center at node i and with radius r, is the set of all the nodes j such that $d(j, i) \leq r$. The *outgoing spherical region* $S \uparrow (i, r)$, with its center at node i and with radius r, is the set of all the nodes j such that $d(i, j) \leq r$.

A key innovation of the proposed algorithm is to make the bounded lag restriction and the minimum propagation delays work in tandem. This reduces the computations needed for determining whether or not a particular event can be processed. The idea can be explained using Figure 4, where a fragment of a large toroidal queuing system network with $N = n^2$ nodes is

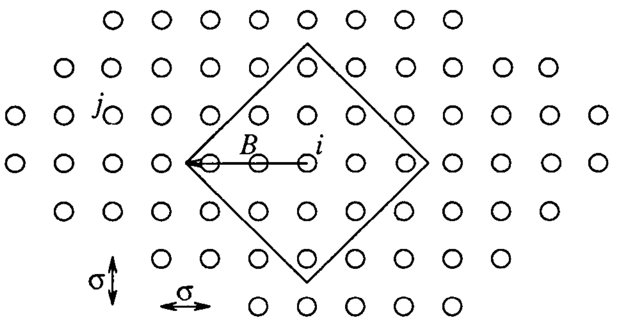


FIGURE 4. Bounded Region of Testing

depicted. Since $d(u, v) = d(v, u)$ for any nodes u and v, one has $S \downarrow (i, r) = S \uparrow (i, r)$. This set will be denoted $S(i, r)$. Figure 4 is scaled so that the distance between two neighbors along a horizontal or a vertical line is equal to the minimum service time σ.

Assume, for simplicity, that all event times are distinct and suppose one wishes to learn if it is possible to process an event e at node i. The time of e must be minimal among the events in Π_i in order to consider e as a candidate for processing. Checking this condition is sufficient in the serial simulation, where only one pool is maintained. However, this test is not sufficient in the distributed simulation. Indeed, an event e' in another local pool Π_j with smaller time $\tau(e') < \tau(e)$ may exist. Processing e', even indirectly, through a chain of intermediate events and pools, may affect event e.

As pointed out in [22], in principle, one should check all other $N - 1$ pools to see if such a "dangerous" event e' exists. Such global screening is unacceptably expensive, however, when N is large. Suppose the algorithm runs with lag B. Then one knows, even without screening, that a node outside region $S(i, B)$, say node j, cannot contain "dangerous" events, because no information from j can reach i early enough to affect e. Thus, the screening area is reduced to the bounded number of nodes lying within $S(i, B)$.

The testing procedure can be as follows: Let

$$T_i \overset{def}{=} \begin{cases} \min_{e \in \Pi_i} \tau(e) & \text{if } \Pi_i \neq \varnothing \\ +\infty & \text{if } \Pi_i = \varnothing \end{cases}$$

If $\alpha(i)$ denotes an estimate of the earliest time when other nodes can affect the history at node i, then

$$\alpha(i) = \min_{\substack{j \in S\downarrow(i,B) \\ j \neq i}} \{d(j, i) + \min\{T_j, d(i, j) + T_i\}\}. \qquad (3.1)$$

Formula (3.1) represents two ways of affecting the history at node i:

1. An event e, $e \in \Pi_j$, $j \neq i$, with time $\tau(e) = T_j$ causes a delivery of an event in Π_i for time $d(j, i) + T_j$.
2. An event e, $e \in \Pi_i$, with time $\tau(e) = T_i$ causes a delivery of an event e' in Π_j, $j \neq i$, for time $d(i, j) + T_i$. Event e', in turn, causes a delivery of an event in Π_i for time $d(j, i) + d(i, j) + T_i$. The net effect of these transactions is that event e, currently existing in Π_i, is reflected back onto Π_i.

Opaque Periods

In the Ising model example, for any two nodes i and j, no matter how far apart, $d(i, j) = 0$. Indeed, for any $\epsilon > 0$, no matter how small, there is a positive probability that a spin change of atom i will affect a spin change of atom j while the difference in time between the changes is less than ϵ.

Thus, the tandem approach seems not to work. However, an efficient distributed simulation of this example is still possible because of the *opaque periods*. An opaque period is a "promise" of a node not to transfer information along a certain outgoing link for a certain period of simulated time.

In the Ising model, after an attempted spin change for node j has been processed, say for simulated time t_j, the next attempt must be scheduled. Since the attempts form a Poisson process with rate λ, the time of the next attempt, say $next\_t_j$, is computed as

$$next\_t_j = t_j - \frac{1}{\lambda} \ln r(\omega), \qquad (3.2)$$

where $r(\omega)$ is an independent realization of a random variable uniformly distributed in (0, 1), and ln is the natural logarithm.

For the time duration $(t_j, next\_t_j)$, no information can propagate via node j, since no change in the state of node j is visible to the other nodes. For example, if j = (22) in Figure 2, and node 21 changes its spin at time t_*, $t_{22} < t_* < next\_t_{22}$, then node 23 has no way to learn about this change from node 22 before time $next\_t_{22}$. But (23) might learn about the change from other nodes, say from (24).

Now let i = (23). Suppose, like node j = (22), all four neighbors j of i have "promised" not to transfer information to i until $next\_t_j$. If

$$next\_t_i < \min_{\substack{j \in neighbors(i) \\ j \neq i}} next\_t_j,$$

then the spin change attempt at node i for time $next\_t_i$ can be safely processed using the current spin values of the four neighbors. This exemplifies the use of opaque periods.

Opaque periods also exist in the queuing system example and in such a context have been used in many algorithms [3, 4, 6, 9, 10, 18, 20, 22], where the property emphasized has been *lookahead* [3], i.e., the *ability to predict* future states of a part of the simulated system, independently of the other parts. We emphasize another side of the same phenomenon, the *inability to communicate through* the given part for the other parts. Figure 5 shows events in the network with graph $1 \rightarrow 2 \rightarrow 3$. Node 2 begins service at the simulated time τ_0 and remains *serving* until time $\tau_0 + t_{service}$. Service duration $t_{service}$ is determined when the event "start service" is processed for simulated time τ_0. The crossed-out circle on axis Π_2 represents this event.

If link $2 \rightarrow 3$ could transfer information during the service, then the best estimate for the earliest time

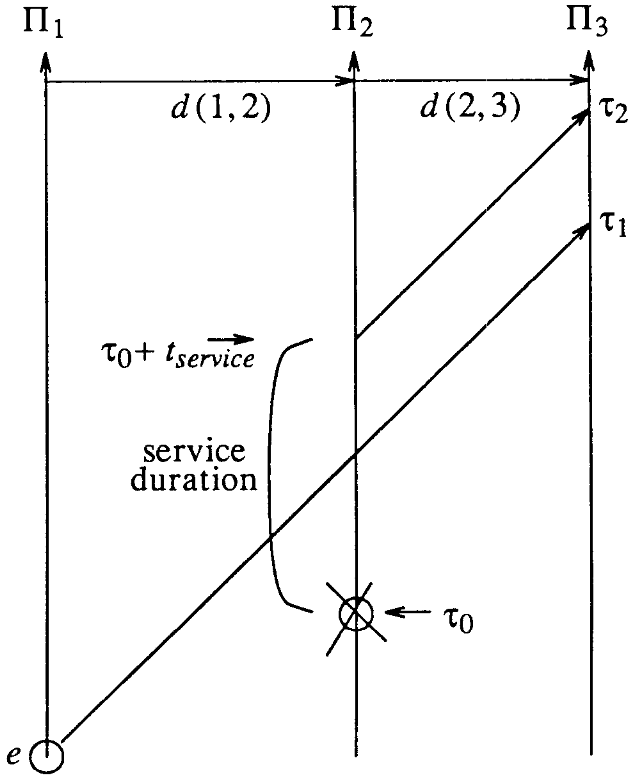


FIGURE 5. Effect of an Opaque Period on Event Propagation

when event e can affect pool Π_3, would be $\tau_1 = \tau(e) + d(1, 3) = \tau(e) + d(1, 2) + d(2, 3)$. However since link $2 \rightarrow 3$ is opaque, that is, unable to transfer information, event e cannot affect pool Π_3 earlier than at time $\tau_2 = \max\{\tau(e) + d(1, 2), \tau_0 + t_{service}\} + d(2, 3)$.

In a typical case, opaque periods cannot be precomputed, unlike the minimum propagation delays. They should instead be determined again at each stage of simulation. Specifically, let t_i be the current simulated time at node i. Then the opaque period for an outgoing link $i \rightarrow j$ has the form (t_i, op_{ij}), where op_{ij} is the end of the opaque period. Since in both considered examples, op_{ij} is independent of j, we call it op_i.

In the Ising model example, op_i is equal to the next scheduled spin change attempt by atom i. Thus, formula (3.2), where $next\_t_j$ should be substituted with op_j, yields op_j. The computation of op_i is performed when a spin change attempt is processed. In the queuing system example, op_i is computed when event $(1, i, t)$ is processed. Namely, after $t_{service}$ is determined by rule 1 in Figure 3, op_i is computed as $op_i = t + t_{service}$.

Both versions of the algorithm in Figure 6, with and without opaque periods, work for the queuing system example. The version with opaque periods sometimes works substantially faster than the one which uses only tandem. In the Ising model simulation, only the version with opaque periods works.

```
1.  while $floor < end\_time$ do {
2.    compute estimate $\alpha(i)$ of the earliest time, when the
        history at node $i$ can be affected by the other nodes;
3.    synchronize;
4.    while $T_i \le floor + B$ and $T_i < \alpha(i)$ do {
5.      $t_i =: T_i$;
6.      process events $e$ with locally minimum time $\tau(e) = T_i$;
        if required, then schedule new events for $\Pi_i$ or other
        $\Pi_j, j \neq i$, and/or delete some events from $\Pi_i$ or other
        $\Pi_j, j \neq i$;
7.      delete the processed events from $\Pi_i$ and compute
        new $T_i$
    };
8.    synchronize;
9.    $floor := \min_{1 \le i \le N} T_i$;   broadcast $floor$ to all nodes;
10.   synchronize;
    }
```

FIGURE 6. A Parallel Event-Driven Simulation Algorithm

The concept of *lookahead ratio* (LAR) is introduced in [6] as mean service time divided by minimal service time. Assuming that service times are random variables (which is not necessarily the case in our queuing system example), LAR expresses the degree of opaqueness. Experiments [6] confirm that efficiency increases with greater LAR.

In other cases, however, LAR may not represent the degree of opaqueness. Without changing distributions, and hence without changing LAR, simply making the queuing model more specific and exploring this specificity, substantially larger opaque periods can be computed. For example, following [19], we can introduce token identities, assume that service durations are independent random variables, and assume FCFS discipline for the queues. (These assumptions are not used in the queuing example in Section 2.) Then the opaque period for a node would include not only the current service period, but also future service periods for all the tokens currently awaiting service in the queue.

4. ALGORITHMS

An algorithm for a distributed event-driven simulation is shown in Figure 6. The algorithm does not assume a specific application; it assumes only that events of arbitrary nature occur at nodes of an arbitrary network. The algorithm maintains N registers $t_1, \ldots t_N$, of simulated time and involves N PEs, one for each Π_i. These PEs execute the program in Figure 6 in parallel. For any $e \in \Pi_i$, time $\tau(e)$ is guaranteed to be not smaller than t_i. At each iteration the algorithm tries to achieve a progress in each local pool and to increase each t_i. During the simulation, t_i does not decrease. The simulation starts with $floor = 0$, $\Pi_i \neq \emptyset$ for at least one i, $\min_{e \in \Pi} \tau(e) \geq 0$, and $t_i = 0$ for all i. Test $T_i \leq floor + B$ at step 4 in Figure 6 assures the bounded lag restriction.

PEi_0 executes this code with index $i = i_0$ starting from step 1. Some computations require cooperation among the PEs. Among these are steps 3, 8, and 10 with statements "synchronize," and step 9 where $\min_{1 \le i \le N} T_i$ is computed and then broadcast. When a PE hits a "synchronize" statement it must wait until all the other $N - 1$ PEs hit the same statement before it may continue. Instructions between these synchronization barriers are not necessarily executed in lockstep. A cooperation between PEs might also be required at step 6, if a PE operates in pools of other PEs and at step 2, as explained below.

Observe that the algorithm is synchronous, whereas the simulated system may be asynchronous. Synchronization in the algorithm prevents undesirable overlaps. For example, computing $\alpha(i)$ at step 2 must not interleave with steps 6 and 7 where some events are being deleted and values T_i are being changed.

One method to compute $\alpha(i)$ at step 2 is using formula (3.1). In this method, PEi communicates with the PEs hosting nodes $j \in S \downarrow (i, B)$. In the queuing system example, if the topology of the host matches that of the target graph G, and if $B > 2\sigma$, then the communication is required not only between the nearest neighbors, but also between distant neighbors. Moreover, the larger the B, the more non-neighbors are involved in communication.

If such a long range communication is undesirable, an alternative method for implementing step 2 can be used. Figure 7 shows two versions of this method. Both versions compute the same $\alpha(i)$ and each so computed $\alpha(i)$ is the same as the corresponding $\alpha(i)$ computed by (3.1), providing that the latter does not exceed $floor + B$. However, unlike algorithm (3.1), these algorithms re-

```
2.1.  $\alpha(i) := +\infty$; $\beta(i) := T_i$;
      $MIN\_\beta := floor$;
2.2.  synchronize;
2.3.  while $MIN\_\beta \le floor + B$ do {
2.4.      $new\_\beta(i) := \min_{j \in neighbors(i), j \ne i} \{d(j, i) + \beta(j)\}$;
2.5.      if($new\_\beta(i) < \alpha(i)$)
              $\alpha(i) := new\_\beta(i)$;
2.6.      synchronize;
2.7.      $\beta(i) := new\_\beta(i)$;
2.8.      synchronize;
2.9.      $MIN\_\beta := \min_{1 \le i \le N} \beta(i)$;
          broadcast $MIN\_\beta$ to all nodes;
2.10.     synchronize;
      }
```

a. A simple stopping criterion

```
2.1.  $\alpha(i) := +\infty$; $\beta(i) := T_i$;
      $MIN\_\beta := floor$; $\alpha\_CHANGED := 1$;
2.2.  synchronize;
2.3.  while $MIN\_\beta \le floor + B$
          and $\alpha\_CHANGED = 1$ do {
2.4.      $new\_\beta(i) := \min_{j \in neighbors(i), j \ne i} \{d(j, i) + \beta(j)\}$;
2.5.      if($new\_\beta(i) < \alpha(i)$) {
              $changed\_\alpha(i) := 1$; $\alpha(i) := new\_\beta(i)$;
          } else $changed\_\alpha(i) := 0$;
2.6.      synchronize;
2.7.      $\beta(i) := new\_\beta(i)$;
2.8.      synchronize;
2.9.      $MIN\_\beta := \min_{1 \le i \le N} \beta(i)$;
          $\alpha\_CHANGED := \max_{1 \le i \le N} changed\_\alpha(i)$;
          broadcast $\alpha\_CHANGED$ and $MIN\_\beta$ to all nodes;
2.10.     synchronize;
      }
```

b. An improved stopping criterion

FIGURE 7. An Alternative Method to Compute $\alpha(i)$

quire only communication between neighboring PEs. All PEs execute the same version, either the one in Figure 7a, or the one in Figure 7b. The convention for interpreting codes in Figure 7 is the same as for the code in Figure 6: PEi_0 executes a code with index $i = i_0$. Auxiliary variables $\beta(i)$, $changed\_\alpha(i)$, and $new\_\beta(i)$ are employed by the algorithms.

To understand these codes, it is useful to view them as a preliminary, coarse simulation [9]. During this pre-simulation, instead of events, estimates of earliest times when events can emerge are propagating across the network. To distinguish iterations 2.4–2.10 of the algorithm in Figure 7 from iterations 2–10 of the algorithm in Figure 6, the latter are called *major* iterations and the former are called *minor* iterations. After k minor iterations, $\beta(i)$ represents an estimate for the earliest time when existing events can affect node i after traversing exactly k links in the graph. Suppose $T_i \le floor + B$. Starting with the value $\beta(i) = T_i$ for $k = 0$, as k increases, $\beta(i)$ increases, being destined to exceed value $floor + B$. However, the increase needs not be monotonic, sometimes $\beta(i)$ may decrease. While $\beta(i)$ is so changing, $\alpha(i)$ is keeping the record of smallest $\beta(i)$ achieved so far, excluding the initial value of $\beta(i)$ at $k = 0$. Thus the direction of change of $\alpha(i)$, unlike the direction of change of $\beta(i)$, can not be reversed.

Now consider the difference between the versions in Figures 7*a* and 7*b*. The version in Figure 7*a* continues the minor iterations until all $\beta(i)$ exceed the value $floor + B$, which is represented by the first test in line 2.3. In the queuing network example, this version terminates after, at most, $\lceil B/\sigma \rceil$ minor iterations. In a general case, this estimate becomes *the largest number m of nodes in a directed path* $i_1 \rightarrow \cdots \rightarrow i_m$ *whose total length* $d(i_1, i_2) + \cdots + d(i_{m-1}, i_m)$ *does not exceed B*. In contrast, the version in Figure 7*b* attempts to decrease this number of minor iterations by using a better stopping criterion at step 2.3. This version capitalizes on the fact that once no decrease of $\alpha(i)$ is detected for all i (step 2.5), $\alpha(i)$ can change no more.

It is easy to prove this fact. Indeed, if $\alpha_{k-1}(i) = \alpha_k(i)$, $i = 1, \ldots N$, where subscript k denotes the minor iteration number, then none of the $\beta_k(i)$, $i = 1, \ldots N$, is smaller than its record low value before minor iteration k. Therefore, none of $new\_\beta_k(i)$ computed at step 2.4 is smaller than its record low value before minor iteration k. By the virtue of step 2.5 of the algorithm, $\alpha_k(i) = \alpha_{k+1}(i)$. This completes the induction step.

Neither formula (3.1) nor the algorithms in Figure 7 uses opaque periods. While it is difficult to incorporate opaque periods in the computations by (3.1), the algorithms in Figure 7 require only a minor modification; thus, step 2.4 in both modified versions reads:

$$new\_\beta(i) := \min_{\substack{j \in neighbors(i) \\ j \ne i}} \{d(j, i) + \max\{\beta(j), op_{ji}\}\}$$

Yet another modification of the algorithm in Figure 6 deals with cases when there are fewer PEs than nodes in the network by having each PE host a subnetwork. A fragment of a large toroidal network is shown in Figure 8, wherein each PE carries a 4 × 4 subnetwork.

The general algorithms presented above can be greatly simplified for the Ising model simulation. Here the bounded lag mechanism is not needed. Therefore, steps 9 and 10 and test $T_i \le floor + B$ at step 4 are eliminated. Computing $\alpha(i)$ reduces to the simple formula

$$\alpha(i) = \min_{\substack{j \in neighbors(i) \\ j \ne i}} op_j.$$

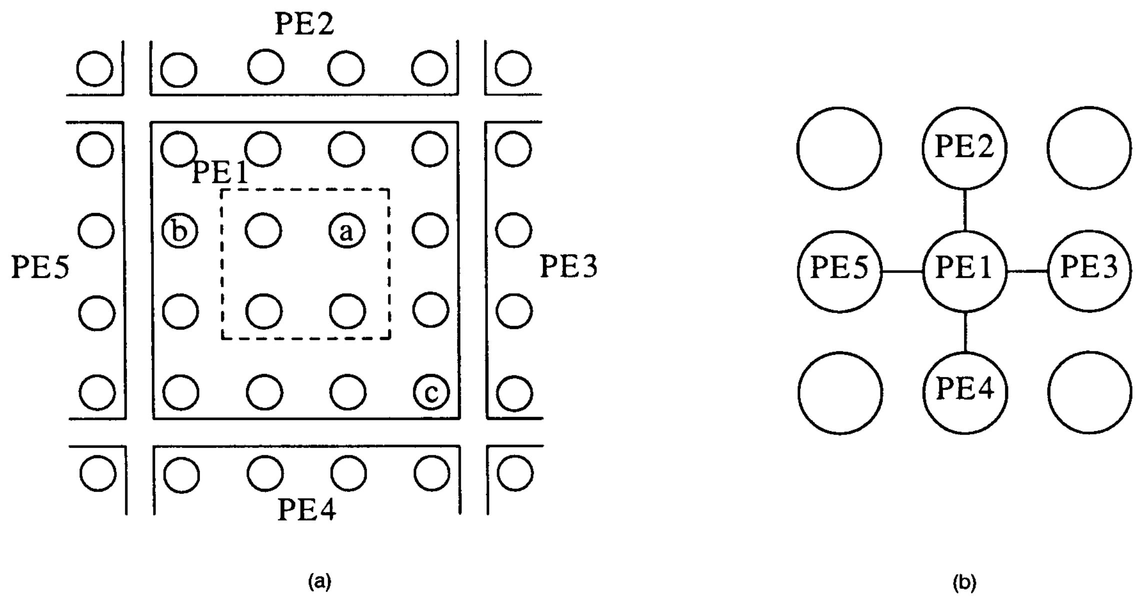


FIGURE 8. Aggregation

Clearly, after processing a current attempted spin change, the current T_i becomes the same as the current op_i. There will be only one event e to process at step 6; no insertion/deletion in another pool Π_j is performed. Finally, with probability 1, no two spin change attempts coincide in time. The latter observation allows one to eliminate global "synchronize" statements and the entire algorithm reduces to the remarkably simple form presented in Figure 9, where $r(\omega)$ and ln are the same as in (3.2).

The algorithm in Figure 9 is very asynchronous: different PEs can execute different steps concurrently and can run at different speeds. A statement "wait_until *condition*," like the one at step 2 in Figure 9, does not imply that the *condition* must be detected immediately after it occurs. In particular, to detect condition

$$T_i \le \min_{\substack{j \in neighbors(i) \\ j \ne i}} T_j,$$

while executing step 2 of this algorithm, a PE can poll the four neighboring PEs in an arbitrary order, one at a time, with arbitrary delays and interruptions, and without any respect to what these PEs are doing meanwhile.

The algorithm in Figure 9 is free from deadlock. Despite its asynchrony, it produces correct simulation histories, that is, those which are statistically equivalent to the histories produced by Metropolis et al. algorithm. Moreover, a history is reproducible, that is, it remains the same, independent of the conditions of run, timing etc., provided that (1) the random number generator always produces the same sequence of r's for each node, and (2) no two neighboring T_i and T_j are equal.

Assuming (2), "less than or equal to" at step 2 in Figure 9 could be replaced with "less than." However, if, in violation of (2), condition

$$T_i = \min_{\substack{j \in neighbors(i) \\ j \ne i}} T_j$$

strikes, then the version with "less than" deadlocks. (A violation of (2) may be caused by a round-off error). The effect of the violation on the version with "less than or equal to" is less severe: just a point of irreproducibility. One may use the synchronous algorithm, if such irreproducibility is intolerable [14].

An aggregation is welcome in the algorithm in Figure 9, since a PE computes very little between communications with the other PEs. In Figure 8, the neighbors of a node hosted by PE1 are nodes hosted by PE2, PE3, PE4, or PE5. PE1 has direct connections with these four PEs (Figure 8*b*). Given node i in the subnetwork of PE1, one can determine with which neighboring PEs communication is required in order to learn the states of

```
1. while $T_i < end\_time$ do {
2.    wait_until $T_i \le \min_{j \in neighbors(i), j \ne i} T_j$;
3.    $s(i) := next\_state(s(neighbors(i)), \omega)$;
4.    $T_i := T_i - \frac{1}{\lambda} \ln r(\omega)$
   }
```

FIGURE 9. One-Spin-per-One-PE Version of an Ising Simulation

the neighboring nodes. Let $W(i)$ be the set of these PEs. Examples in Figure 8 are: $W(a) = \emptyset$, $W(b) = \{\text{PE5}\}$, $W(c) = \{\text{PE3, PE4}\}$.

Figure 10 presents an aggregated variant of the algorithm in Figure 9. PEI hosts the Ith subnetwork and maintains the local time register T_I, $I = 1, 2, \ldots, (n/m)^2$. PE$I_0$ simulates the evolution of its subnetwork using the algorithm presented in Figure 10 with $I = I_0$. Local times T_I maintained by different PEs might be different. However, within a subnetwork the simulation is serial, like in the standard algorithm, and the local time for all atoms in subnetwork I is the same, namely T_I. In $(m - 2)^2$ cases out of m^2, step 3 of the algorithm in

```
1. while $T_I < end\_time$ do {
2.     select a node $i$ in the subnetwork $I$ uniformly at random;
3.     wait_until $T_I \le \min_{J \in W(i)} T_J$;
4.     $s(i) := next\_state\,(s(neighbors(i)), \omega)$;
5.     $T_I := T_I - \frac{1}{\lambda m^2} \ln r(\omega)$
   }
```

FIGURE 10. Many-Spins-per-One-PE Version of an Ising Simulation

Figure 10 involves no busy wait, since $W(i) = \emptyset$. The atoms i such that $W(i) = \emptyset$ constitute the dashed square in Figure 8.

5. PERFORMANCE OF SIMULATION

First, consider the Ising simulation algorithm in Figure 9. The performance of this algorithm will be assessed for the case when PEs execute at the same speed and synchronously as if the execution were organized in iterations. At step 2, PEi would start the iteration by first polling each of its neighbors and then testing the T_i versus the minimum of the other four T_j's. It would then complete the iteration by executing steps 3 and 4, if the test succeeded, or by waiting for the next iteration, if the test failed. This is a possible but not obligatory timing arrangement for executing the algorithm in Figure 9.

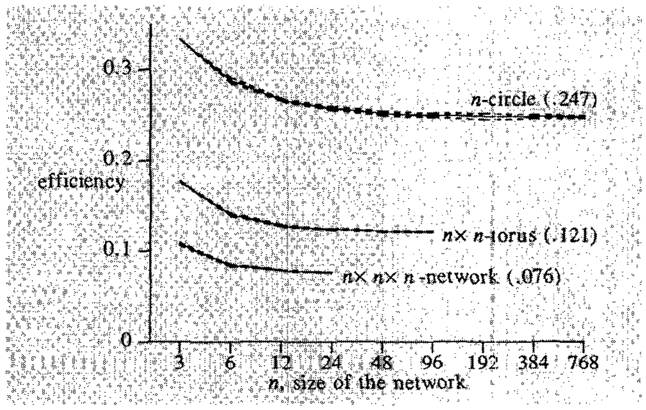
It is interesting to know what fraction of all N PEs succeeds the test at step 2. Those succeeding PEs make progress and perform useful work while the others are idle. Clearly, the law of spin updates implemented at step 3 in Figure 9 is irrelevant in answering this question.

Using these rules, a serial algorithm for updating T_i's in iterations is exercised for different system sizes n and three different dimensions: for an n-circle network, for an $n \times n$ toroidal network (as in the model discussed in section 2), and for an $n \times n \times n$ network.

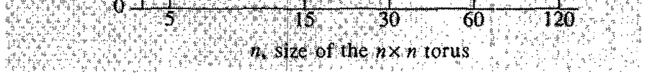
The results of these experiments are given in Figure 11a, where the efficiency is calculated as the average fraction of those T_i's being updated at an iteration. Each solid line in Figure 11a is enclosed between two dashed lines. The latter represent 99.99 percent Student's confidence intervals constructed using several simulation runs, which were parametrically equal but fed with different random sequences.

No analytical theory is available for predicting the fraction of non-idle PEs, or even for proving its separation from zero when $n \rightarrow +\infty$. However it is clear from Figure 11a, that at least 12 percent of $n^2 = N$ PEs are doing useful work. With such efficiency, the speed-up is about $0.12 \times N$; for $N = 2^{14}$ the speed-up is more than 1900.

These numbers were confirmed in an actual experiment performed on $2^{14} = 128 \times 128$ PEs of a Connection Machine (a quarter of the full computer was available). This SIMD computer, equipped with the language *LISP, appeared well-suited for synchronous execution of the one-spin-per-one-PE algorithm in Figure 9 on a toroidal network. Note that the so-called multi-spin algorithms [5] update faster, but simulate a different, *syn-*



(a)



(b)

FIGURE 11. Performance of the Ising Model Simulation: (a) One spin per one PE; (b) many spins per one PE

chronous Ising model which is trivial for parallelization. In contrast, Metropolis et al. algorithm, which is simulated in the experiments described here, was previously believed inherently serial.

The 12 percent efficiency in the one-spin-per-one-PE experiments could be greatly increased by aggregation. The many-spins-per-one-PE algorithm in Figure 10 was implemented as a parallel C-program for a Balance computer, which is a shared memory MIMD bus machine. The $n \times n$ network was split into $m \times m$ subnetworks as shown in Figure 8, where n is an integer multiple of m. The computer has 30 PEs, therefore the experiments could be performed only with $(n/m)^2 =$ 1, 4, 9, 16, and 25 PEs for different n and m.

Along with these experiments, a simplified serial model, similar to the one of the one-spin-per-one-PE case, was run. As in the previous case, this simplified model simulates a possible but not obligatory timing arrangement for executing the real algorithm, here the algorithm in Figure 10. Figure 11*b* shows excellent agreement between performances of the two models for the aggregated Ising simulations. The efficiency presented in Figure 11*b* is computed by the formula

$$\text{efficiency} = \frac{\text{serial execution time}}{\text{number of PEs} \times \text{parallel execution time}} \tag{5.1}$$

The parallel speed-up can be found as efficiency × number of PEs. Efficiency is 0.66 for 25 PEs simulating a 120 × 120 Ising model; hence, the speed-up is greater than 16. For the currently unavailable sizes, when 10^4 PEs simulate a $10^4 \times 10^4$ network, the simplified serial model predicts efficiency at about 0.8 and speed-up at about 8000.

We shall now outline the proof of the $1/\log N$ scalability of the general algorithm in Figure 6. (A detailed proof is given in [15].) It is possible to design the host computer so that each "synchronize," and each computation and broadcast of minimum or maximum, costs $O(\log N)$. Under certain assumptions of network sparsity (e.g., assumption 2 in the Appendix), computation of $\alpha(i)$ involves $O(1)$ synchronizations and costs $O(1)$ instructions between synchronizations. This holds for all three versions of the algorithm, whether using formula (3.1), or the method in Figure 7*a* or 7*b*. Under the assumption that the number of pending events in each Π_i is bounded from above by a constant independent of N and i, the cost of pool manipulations needed for processing one event is $O(1)$. If we further assume that, independent of i and the major iteration, there are only $O(1)$ events to process in each Π_i at each major iteration, we come to the overall cost of $O(N \log N)$ instructions for one major iteration of the algorithm in Figure 6.

To evaluate performance, we need an estimate of the number of events processed during a major iteration. In principle, the simulated system may exhibit low activity, as in the case of a single token traveling in a large network. If this is not the case, and the system exhibits a reasonably dense activity in space and time, then at least an order of N events is processed at each major iteration. The proof of the latter statement is outlined in the Appendix, where, in particular, the "reasonably dense activity" assumption is stated as a requirement that the number of nodes which have pending events e, satisfying $\tau(e) \leq \mathit{floor} + B$, be, on average, not less than of an order N. Hence the average cost of processing one event is $O(\log N)$, which also means at least an order of $1/\log N$ scalability.

The argument formulated above for the case of N PEs, applies to cases where there are fewer PEs than nodes N, provided that the maximal number of nodes per PE is $O(1)$ when $N \rightarrow \infty$.

In a serial event-list algorithm, the length of the list would be an order of N. Best list manipulation techniques require on average an order of $\log N$ instructions for performing insertions and deletions needed for one event processing. Hence, a conventional serial algorithm spends about the same number of instructions processing one event, as one PE in a parallel processor. Observe that such a conventional algorithm does not use minimum propagation delays or opaque periods. Since there are N PEs running, the speed-up would be an order of N.

A necessary first step in writing a parallel program is to write a serial program. Such a serial code, which simulates an $n \times n$ queuing model, is now operational. In the code, each parallel step of the algorithm in Figure 6 is represented as a serial loop, node number i being the loop index. Minimization is done in a usual way, so it costs an order of N instructions.

The steps which in the parallel algorithm cost an order of $\log N$ to each PE, are either eliminated or cost only an order of N in the serial algorithm. Therefore, each major iteration should cost at most an order of N, and the per-event cost should be $O(1)$. The latter statement is tested using the following procedure. The model is exercised with various network sizes n, but with other fixed parameters μ, σ, p, B, and *end_time*. Only those n which are integer multiples of 4 are taken. The network contains $(n/4)^2$ sources, located at nodes $(4u, 4v)$, $u, v = 1, 2 \ldots, (n/4)$. As explained in section 2, the source buffers are full to capacity $(+\infty)$. There is no sink, but the positivity of p assures that the number of tokens circulating in the network saturates after a certain transient period. For all experiments the saturation occurs before the first 1000σ simulated seconds elapse; the timing and other data are recorded during the immediately following interval of length 1000σ, so that *end_time* $= 2000\sigma$.

A series of execution times is obtained, and each time is divided by n^2, the number of nodes. The results for the version of the algorithm where α are computed as in Figure 7*a*, are presented in Figure 12. Because of execution time variability, the same run is repeated several times. As in the Ising model experiments, the dashed curves in Figure 12 represent 99.99 percent Student's confidence intervals constructed from these samples.

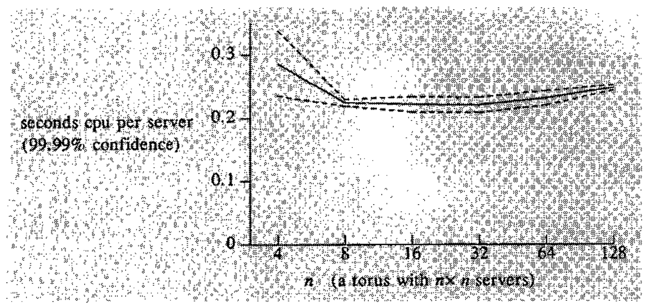


FIGURE 12. Performance of the Queuing System Simulation

Figure 12 shows very little increase of the per-node CPU time, despite a substantial increase in the problem size. Since the number of sources rises proportionally to the problem size, it is reasonable to presume the number of events per node to be independent of system size in these experiments. This agrees with the conjectured per-event performance $O(1)$. The results are similar for the other two versions of the algorithm.

6. CONCLUSION: OTHER EXAMPLES, SHORTCOMINGS, FURTHER RESEARCH

Only regular, highly symmetrical networks with simple communication structures (i.e., having only one token type, atoms with only two states, infinite buffers, etc.) were chosen for presentation in this article. Irregularity may lead to two problems: (1) a more complex map from the simulated system into the abstract structure of the algorithm (more event types, more complex processing rules, and opaque period computations), (2) load unbalance during simulation.

The first problem should be addressed by considering a wide class of applications. An example of a token transport network simulation with *finite buffers* and nonzero *transmission durations* has been programmed [15]. An interesting phenomenon in this example is that minimum propagation delays and opaque periods stem not only from nonzero service durations but also from transmission durations. The latter contribute to the propagation delays in the counter-traffic direction, e.g., while a token travels from node i to node j, some associated events propagate from j to i.

In an example of a timed logic simulation currently under investigation there are readily defined propagation delays, but the determination of opaque periods appears somewhat different from that of the queuing systems and the Ising model. For example, an *OR*-gate with signal 1 at one input is guaranteed to remain opaque until this signal changes.

By choosing in this article the symmetrical networks, we isolated the problem of efficient parallel coordination from the problem of load balancing. The issue of load unbalance, which is of a major practical importance, is of a different nature and, we believe, should be solved by different methods [21].

The speed of Ising simulations can be further increased [14] by incorporating a serial algorithm [2]. The latter works faster than the standard algorithm [16] by avoiding processing unsuccessful spin change attempts. In the Ising model, algorithms [7, 11] would also capitalize on the unsuccessful spin change attempts by guessing them correctly, when the neighbors' spins are not yet known. It would be interesting to compare the performance of the two approaches.

The basic paradigm of the Ising simulation can be applied to a wider class of applications than one might think. These include cellular arrays [13], timed Petri nets, asynchronous neural networks, and communication networks. For example, in a telephone network, calls arriving on a particular route, say a pair of nodes i and j, form a Poisson process. To figure out the new state of the route after the arrival, the states of the incident routes must be known. This situation is expressed by equation (2.1), only instead of a graph one should consider a hypergraph. A node of this hypergraph would be the set of original nodes constituting a route. The algorithm performance would depend on the number of neighbors, i.e., incident routes. For the one-spin-per-one-PE Ising simulation, Figure 11*a* shows that performance quickly degrades as the dimension increases: a PE waits for more neighbors at step 2 of the algorithm in Figure 9 as the local degree of the network increases. This shortcoming can be compensated by aggregation. The success in the aggregation depends on the graph sparsity. For example, if the diameter of the graph is small, aggregation would not result in a substantial reduction of the wait.

To succeed in applying the tandem approach, global topological properties such as diameter are also important. Examples of particularly bad topologies for the proposed algorithms of simulation are a star, where $N - 1$ peripheral nodes are connected to a central node, and a fully connected graph. It becomes clear from the queuing system example in section 2, that propagation delays do not necessarily mean the physical signal propagation delays between nodes. The latter may be negligibly small and assumed to be zero in a model. Yet the minimum propagation delays may be substantial because of known positive lower bounds on the service times.

Positive lower bounds on durations of various activities in practical systems can be found, perhaps, more often, than is discussed in theoretical papers. These papers usually assume exponential distributions, which facilitate finding analytical solutions to the models. However, in some applications the distributions are exponential, e.g., if a queuing theorist wishes to verify his theory by a computer simulation. In such cases the precomputed lower bound is zero, and the tandem described earlier seems not to work.

However, a simple modification enables the tandem to work, if we can compute positive *dynamic* propagation delays, instead of precomputed (zero) *static* propagation delays. In stochastic network simulations, positive dynamic propagation delays are zero. For example, we can take a restricted stochastic version of the general token transport model in section 2, and assume the service durations to be random and independent. Because of the independence, we can presimulate durations of all the services by server i that belong to the future, not yet created history, without respect to what the other servers will be doing during this future history. (A similar presimulation is used in [19].) Specifically, the PE hosting server i can presimulate enough service durations to cover the interval $[\max(floor, t_i), floor + B]$ (based on the worst case possibility that the server is never *idle*). Then, for any neighbor j, the PE sets $d(i, j)$ to be the minimum of these durations. At each major iteration, as *floor* increases, and the time interval $[\max(floor, t_i), floor + B]$ changes, the PE updates the set of presimulated durations as well as their minimum $d(i, j)$. The dynamic $d(i, j)$ can be substantially larger than its static lower bound σ. Formula (3.1) becomes inconvenient in this modification, because the dynamic delay $d(i, j)$ is readily available only for neighboring i and j. However, the alternative algorithms in Figure 7 work without change. A similar technique works well in more complex stochastic simulation models, e.g., in models with preemptive services and with several classes of tokens.

In some problems, rather than just picking a random activity duration, the simulator might be required to mimic the simulated activity. A typical example is an instruction-level simulation of program executions, where the host executes each instruction of the simulated program on behalf of the simulated system. If, in such models, sufficiently large minimum propagation delays $d(i, j)$ are hard to determine, statically or dynamically, then the proposed algorithm can be combined with a roll-back algorithm, e.g., with Time Warp [11]. In the combined algorithm, the minimum propagation delays $d(i, j)$ are substituted by surrogate values. The surrogate $d(i, j)$ accounts not only for the delays in propagating events from i to j but also for the frequency of sending events over link $i \rightarrow j$. Since the lower bound $d(i, j)$ is not guaranteed, occasionally a delay in propagating an event from i to j can be smaller than $d(i, j)$. Therefore, the roll-back might be needed. However, unlike the "raw" roll-back [11], the combined roll-back/bounded-lag algorithm, controls the frequency of roll-backs and limits their harmful cascading by adjusting $d(i, j)$ and the lag bound B. This method of simulation is presently under study.

Acknowledgments. I thank the personnel of the Thinking Machine Corp. for their kind invitation and help in debugging and running the parallel program on one of their computers. Particularly, the help of Mr. Gary Rancourt and Mr. Bernie Murray was invaluable.

APPENDIX:

On Average, at Least an Order of N Events are Processed at a Major Iteration

An event e is said to be *within the B-horizon*, if $\tau(e) \leq floor + B$. Given the minimum-propagation-delays construct and using the fact that the algorithm in Figure 6 maintains the bounded lag property with parameters B, one easily proves the following

Proposition 1. If pool Π_j contains at least one event within the B-horizon, then there exists at least one node $i \in S\downarrow(j, B)$ such that test $T_i < \alpha(i)$ (at step 4 of the algorithm in Figure 6) succeeds.

With Proposition 1, one then establishes

Proposition 2. The number of events processed at iteration k of the algorithm in Figure 6 is at least

$$\frac{\text{number of pools with events within the } B\text{-horizon at iteration } k}{\max_{1\leq i\leq N} |S\uparrow(i, B)|}. \tag{A.1}$$

If the density of events is too low or graph G is not sufficiently sparse, then there is no hope that many events will be processed at an iteration. Thus, two assumptions are introduced:

Assumption 1. The numerator in (A.1) or, perhaps, its amortized estimate (over the simulation run) is at least an order of N.

Assumption 2. There exists a constant D, $0 < D < +\infty$, independent of N, such that $\max_{1\leq i\leq N} |S\uparrow(i, B)| \leq D$.

Combining Assumptions 1 and 2 and Proposition 2, one proves that at least an order of N events are processed, on the average, at a major iteration of the algorithm in Figure 6.

REFERENCES

1. Binder, K. (ed.). *Monte Carlo methods is statistical physics*, Springer-Verlag, New York, N.Y., 1986.
2. Bortz, A.B., Kalos, M.H., and Lebowitz, J.L. A new algorithm for Monte-Carlo simulation of Ising spin systems, *J. Comp. Physics, 17*, 1 (Jan. 1975), 10–18.
3. Chandy, K.M., and Misra, J. Distributed simulation: A case study in design and verification of distributed programs, *IEEE Trans. Software Eng., SE-5*, 5 Sept. 1979, 440–452.
4. Chandy, K.M., and Misra, J. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM 24*, 3 (Apr. 1981), 198–206.
5. Friedberg, R., and Cameron, J.E. Test of the Monte Carlo method: Fast Simulation of a Small Ising Lattice, *J. Chem. Physics 52*, 12 (1970), 6049–6058.
6. Fujimoto, R.M. Performance measurements of distributed simulation strategies. In *Proceedings of the 1988 SCS Multiconference* (San Diego, Feb. 3–5, 1988). Simulation Series, SCS *19*, 3, 14–20.
7. Gafni, A., Berry, O., and Jefferson, D. Optimized virtual synchronization. In *Proceedings of the 2d International Workshop on Applied Mathematics and Performance/Reliability Models* (Rome, Italy, May 25–29, 1987). Univ. of Rome II (1987), 229–244.
8. Glauber, R.J. Time-dependent statistics of the Ising model. *J. Math. Physics 4*, 2 (1963), 294–307.
9. Groselj, B., and Tropper. C. Pseudosimulation: An algorithm for distributed simulation with limited memory. *Int. J. Parallel Programming 15*, 5 (1987), 413–456.
10. Holmes, V. *Parallel algorithms on multiple processor architectures*, Ph.D. dissertation, Comp. Science Dept., Univ. Texas at Austin, (1978).
11. Jefferson, D.R. Virtual time. *ACM Trans. Prog. Lang. and Syst. 7*, 3 (July 1985), 404–425.
12. Jones, D.W., Concurrent simulation: an alternative to distributed simulation. In *Proceedings of the 1986 Winter Simulation Conference* (Washington, D.C., Dec. 8–10, 1986). 417–423.
13. Lubachevsky, B.D. Efficient parallel simulations of asynchronous cellular arrays. *Complex Systems 1*, 6 (Dec. 1987), 1099–1123.
14. Lubachevsky, B.D. Efficient parallel simulations of dynamic Ising spin systems. *J. Comp. Physics 75*, 1 (Mar. 1988), 103–122.
15. Lubachevsky, B.D. Bounded lag distributed discrete event simulation, Bell Laboratories Tech. Rep., 1986, submitted for publication, (shortened version). In *Proceedings of the 1988 SCS Multiconference*. Simulation Series, SCS, *19*, 3, 183–191.

(*continued on p. 131*)

Finally, the cases when X1 has exactly 3 ones in its 4-neighborhood are

$$\begin{array}{ccc} C & X2 & \\ X0 & X1 & 1 \\ & 1 & \\ & (a) & \end{array} \qquad \begin{array}{cccc} & 1 & D & \\ X0 & X1 & X2 & J \\ & 1 & E & \\ & (b) & & \end{array} \qquad \begin{array}{ccc} & 1 & \\ X0 & X1 & 1 \\ C & X2 & \\ & (c) & \end{array}$$

and the 90 deg. rotations of these. For cases (a) and (c) $C = 1$ to avoid 4-connecting X0 and X2 in S'. This implies that X2 is not deletable by Lemma 1. For case (b) $D = E = 1$ to allow X1 to be deletable and $J = 0$ to allow X2 to be deletable. C3 will be violated for these cases if X0 and J are not already 4-connected in S'. This can occur as shown here

$$\begin{array}{ccccc} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{array}$$

where FP connects the hole in the object to the background. C3 would be maintained for this case if additional conditions on FP guarantee that X1 and X2 were not both deletable which completes the proof for the C3 part of Theorem 1. □

REFERENCES

1. Arcelli, C., Cordella, L.P. and Levialdi, S. Parallel thinning of binary pictures. *Electronics Letters 11*, 7 (Apr. 1975), 148–149.
2. Golay, M.J.E. Hexagonal parallel pattern transformations. *IEEE Trans. Comput. C-18*, 8 (Aug. 1969), 733–740.
3. Guo, Z. and Hall, R.W. Parallel thinning with two-subiteration algorithms. Submitted for publication.
4. Holt, C.M., Stewart, A., Clint, M., and Perrott, R.H. An improved parallel thinning algorithm. *Comm. ACM 30*, 2 (Feb. 1987), 156–160.
5. Lü, H.E. and Wang, P.S.P. A comment on "A fast parallel algorithm for thinning digital patterns." *Comm. ACM 29*, 3 (Mar. 1986), 239–242.
6. Preston, K. and Duff, M.J.B. *Modern Cellular Automata.* Plenum, New York, 1984.
7. Rosenfeld, A. A characterization of parallel thinning algorithms. *Information and Control* 29, 3 (Nov. 1975), 286–291.
8. Rosenfeld, A. and Kak, A. *Digital Picture Processing.* vol. 2, Academic Press, New York, 1982.
9. Rutovitz, D. Pattern recognition. *J. Royal Statist. Soc. 129, Series A* (1966), 504–530.
10. Stefanelli, R. and Rosenfeld, A. Some parallel thinning algorithms for digital pictures. *J. ACM 18*, 2 (Apr. 1971), 255–264.
11. Wang, P.S.P. and Zhang, Y.Y. A fast serial and parallel thinning algorithm. In *Proceedings of the Eighth European Meeting on Cybernetics and Systems Research 86* (Vienna, Austria, April 1–4, 1986), R. Trappl ed. 1986, pp. 909–915.
12. Zhang, T.Y., and Suen, C.Y. A fast thinning algorithm for thinning digital patterns. *Comm. ACM 27*, 3 (Mar. 1984), 236–239.

CR Categories and Subject Descriptors: I.4 **[Image Processing]**; I.5.2 **[Pattern Recognition]**: Design Methodology—*pattern analysis;* I.5.4 **[Pattern Recognition]**: Applications—*computer vision*

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Parallel algorithms, preservation of connectivity, thinning binary patterns, two-subiteration algorithms.

ABOUT THE AUTHOR:

RICHARD W. HALL is an Associate Professor of Electrical Engineering at the University of Pittsburgh. His current research interests include computer vision and parallel algorithms and architectures for visual information processing. Author's Present Address: Department of Electrical Engineering, University of Pittsburgh, Pittsburgh, PA 15261.

Lubachevsky *(continued from p. 123)*

16. Metropolis, N., et al. Equation of state calculations by fast computing machines, *J. Chem. Physics, 21*, 6 (June 1953), 1087–1092.
17. Misra, J. Distributed discrete-event simulation. *Comput. Surv. 18*, 1 (1986), 39–65.
18. Nicol, D.M. *Synchronizing network performance.* M.S. dissertation, School of Eng. and Appl. Science, Univ. of Virginia, (1984).
19. Nicol, D.M. Parallel discrete-event simulation of FCFS stochastic queuing networks. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming Experience in Applications, Languages, and Systems,* (New Haven, Conn., July 1988).
20. Nicol, D.M., and Reynolds, P.F. Problem oriented protocol design. In *Proceedings of the 1984 Winter Simulation Conference* (Dallas, Tex., Dec. 1984). 471–474.
21. Nicol, D.M., and Reynolds, P.F. An optimal repartitioning decision policy. In *Proceedings of the 1985 Winter Simulation Conference* (San Francisco, Calif., Dec. 1985). 493–497.
22. Peacock, J.K., Wong, J.W., and Manning, E.G. Distributed simulation using a network of processors, In *Proceedings of the 3d Berkeley Workshop on Distributed Data Managements and Comp. Networks* (1978) and *Computer Networks 3*, 1 (1979).
23. Reed, D.A., Malony, A.D., and McCredie, B.D. Parallel discrete event simulation using shared memory, *IEEE Trans. Software Eng., 14*, 4 (1988), 541–553.
24. Sokol, L.M., Briscoe, D.P., and Wieland, A.P. MTW: a strategy for scheduling discrete simulation events for concurrent execution. In *Proceedings of the 1988 SCS Multiconference* (San Diego, Calif., Feb. 3–8, 1989). Simulation Series, SCS, *19*, 3, 34–42.

CR Categories and Subject Descriptors: C.3 **[Special Purpose and Application-Based Systems]**; I.6.1 **[Simulation and Modeling]**: Simulation Theory

General Terms: Design, Experimentation

Additional Key Words and Phrases: Discrete-Event simulation, event-driven simulation, parallel processing, Ising model

ABOUT THE AUTHOR:

BORIS D. LUBACHEVSKY received his *diploma* in mathematics (equivalent M.S.) from Leningrad University in 1971 and the *candidate* degree (equivalent Ph.D.) in Computer Science in 1977 from Tomsk Polytechnical Institute (USSR). He is a member of the Technical Staff in AT&T Bell Laboratories, and author of a number of papers on efficient parallel programming techniques and on distributed simulations [13–15]. His research interests include parallel programming and simulation techniques. Author's present address: Boris D. Lubachevsky, AT&T Bell Laboratories, 600 Mountain Ave., Rm. 2C-121, Murray Hill, NJ 07974.